(19)日本国特許庁（ＪＰ）　(12)公開特許公報（Ａ）　(11)特許出願公開番号
特開2001－101011
（P2001－101011A）
(43)公開日　平成13年４月13日(2001.4.13)

(51)Int.Cl.⁷	識別記号	ＦＩ	テーマコード（参考）
Ｇ０６Ｆ　9/45		Ｇ０６Ｆ　9/44	３２２Ａ ３２２Ｂ

審査請求　未請求　請求項の数18　ＯＬ　外国語出願　（全 39 頁）

(21)出願番号　特願2000－221307(P2000－221307)
(22)出願日　平成12年７月21日(2000.7.21)
(31)優先権主張番号　６０／１４５１３６
(32)優先日　平成11年７月21日(1999.7.21)
(33)優先権主張国　米国（ＵＳ）
(31)優先権主張番号　０９／５４０５７６
(32)優先日　平成12年３月31日(2000.3.31)
(33)優先権主張国　米国（ＵＳ）

(71)出願人　591064003
サン・マイクロシステムズ・インコーポレーテッド
SUN MICROSYSTEMS, INCORPORATED
アメリカ合衆国　94303　カリフォルニア州・パロ　アルト・サン　アントニオ　ロード・901
(74)代理人　100096817
弁理士　五十嵐　孝雄　（外２名）

最終頁に続く

(54)【発明の名称】　デバッガプロトコルジェネレータ

(57)【要約】
【課題】　ＪＤＷＰ通信プロトコル等の仕様に対して共に互換性を有するようなフロントエンドコードとバックエンドコードとを、自動的に生成するための方法を提供する。
【解決手段】　先ず、フロントエンドコードとバックエンドコードの間における通信プロトコルの記述を含む、詳細なプロトコル仕様を記述する。次に、仕様の構文解析を行うコードジェネレータに、詳細な仕様を入力する。次いで、形式仕様からフロントエンドコードを自動的に生成し、Ｊａｖａ（登録商標）プログラミング言語等の第１のコンピュータ言語で記述する。そして、形式仕様からバックエンドコードを生成し、Ｃ等の第２のコンピュータ言語で書き込む。

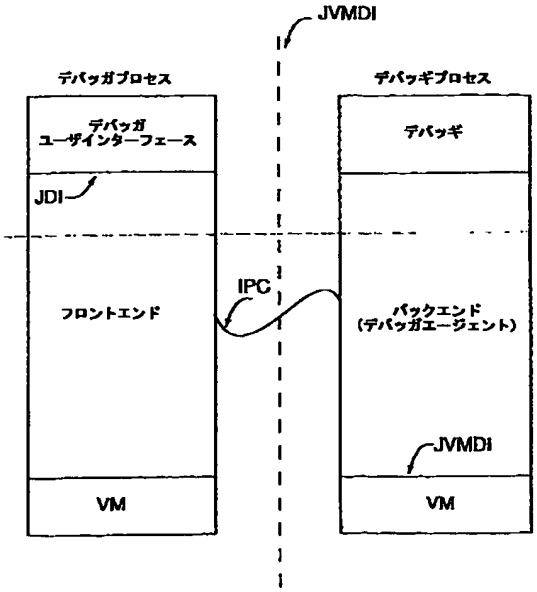

Figure 2a

1

【特許請求の範囲】
【請求項１】　形式仕様と、フロントエンドデバッガプログラムと、デバッギシステムと連結しているバックエンドデバッガプログラムと、の間において互換性を確保するための方法であって、
形式仕様をコードジェネレータに入力し、
コードジェネレータを利用して前記形式仕様を構文解析し、
前記形式仕様からフロントエンドデバッガプログラム部分を生成し、
前記形式仕様からバックエンドデバッガプログラム部分を生成すること、を備えた方法。
【請求項２】　請求項１記載の方法であって、
前記フロントエンドデバッガプログラムは第１の仮想マシン上を走る、方法。
【請求項３】　請求項２記載の方法であって、
前記フロントエンドデバッガプログラム部分はＪａｖａプログラミング言語コードを有する、方法。
【請求項４】　請求項１記載の方法であって、
前記バックエンドデバッガプログラムは、第２の仮想マシンに対して制御および通信を直接行う、方法。
【請求項５】　請求項４記載の方法であって、
前記バックエンドデバッガプログラム部分はＣ言語コードである、方法。
【請求項６】　請求項１記載の方法であって、
前記形式仕様はＪａｖａデバッグワイヤプロトコル仕様である、方法。
【請求項７】　請求項６記載の方法であって、さらに、
前記プロトコル仕様について人が判読可能な記述を含むＨＴＭＬコードを生成すること、を備えた方法。
【請求項８】　請求項１記載の方法であって、さらに、
前記フロントエンドプログラムと前記バックエンドプログラムの間における、前記形式仕様によって定義された通信プロトコルを、使用可能にすること、を備えた方法。
【請求項９】　請求項１記載の方法であって、
前記形式仕様は宣言型の言語で書かれている、方法。
【請求項１０】　請求項８記載の方法であって、
前記通信プロトコルはＪａｖａデバッグワイヤプロトコルである、方法。
【請求項１１】　請求項８記載の方法であって、さらに、
前記形式仕様によって定義された前記通信プロトコルについて人が判読可能な記述を含むＨＴＭＬコードを、前記形式仕様から生成すること、を備えた方法。
【請求項１２】　共に通信プロトコルに対して互換性を有するようなフロントエンドデバッガインターフェースコードとバックエンドデバッガエージェントインターフェースコードとを、自動的に生成するための方法であって、

2

前記フロントエンドデバッガコードと前記バックエンドデバッガエージェントコードの間における通信プロトコルの記述を含む、形式仕様ファイルを書き、
前記形式仕様ファイルをコードジェネレータに入力し、
前記コードジェネレータを利用して前記形式仕様を構文解析し、
前記形式仕様から前記フロントエンドデバッガインターフェースコードを生成し、
前記形式仕様から前記バックエンドデバッガエージェントインターフェースコードを生成すること、を備えた方法。
【請求項１３】　請求項１２記載の方法であって、
前記フロントエンドデバッガインターフェースコードはＪａｖａコードであり、前記バックエンドデバッガエージェントインターフェースコードはＣコードであり、前記形式仕様は使用言語である、方法。
【請求項１４】　請求項１３記載の方法であって、
前記通信プロトコルはＪａｖａデバッグワイヤプロトコルである、方法。
【請求項１５】　共に通信プロトコルに対して互換性を有するようなフロントエンドデバッガインターフェースコードとバックエンドデバッガインターフェースコードとを、自動的に生成するためのコンピュータプログラムコードを含む、コンピュータ読み取り可能媒体であって、
形式プロトコル仕様をコードジェネレータに入力するためのコンピュータプログラムコードと、
前記コードジェネレータを利用して前記形式プロトコル仕様を構文解析するためのコンピュータプログラムコードと、
前記仕様からフロントエンドデバッガインターフェースコンピュータコードを生成するためのコンピュータコードと、
前記仕様からバックエンドデバッガインターフェースコンピュータコードを生成するためのコンピュータコードと、を備えた媒体。
【請求項１６】　請求項１５記載の媒体であって、さらに、
前記通信プロトコルについて人が判読可能な記述を含むＨＴＭＬコードを生成するためのコンピュータコードと、を備えた媒体。
【請求項１７】　請求項１６記載の媒体であって、
前記通信プロトコルはＪａｖａデバッグワイヤプロトコルである、媒体。
【請求項１８】　共に通信プロトコルに対して互換性を有するようなフロントエンドデバッガインターフェースコードとバックエンドデバッガインターフェースコードとを、自動的に生成するためのコンピュータシステムであって、
プロセッサと、

前記プロセッサ上で作動して、形式通信プロトコル仕様を読み込み、前記仕様を構文解析し、前記仕様に対して完全に互換性を有するとともに、互いの互換性も有するフロントエンドデバッガインターフェースコードとバックエンドデバッガインターフェースコードとを生成する、コンピュータプログラムと、を備えたシステム。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】この発明は、概してコンピュータソフトウェアの分野に関し、特に、形式仕様（formal specification）からソフトウェアコンポーネントを生成するためのプロトコル生成ソフトウェアに関する。

【0002】

【従来の技術】Ｊａｖａ（登録商標）デバッグワイヤプロトコル（ＪＤＷＰ：Java Debug Wire Protocol）（Ｊａｖａ〔登録商標〕および関連のマークは、サンマイクロシステムズの登録商標である）は、デバッガアプリケーションとＪａｖａ仮想マシン（ターゲットＶＭ）の間で通信を行うためのプロトコルである。ＪＤＷＰを実装することにより、デバッガは同一または遠隔のコンピュータ上の異なるプロセスにおいて動作可能となる。Ｊａｖａ（登録商標）のプログラミングアプリケーションは、種々様々な異なるハードウェアプラットフォームやオペレーティングシステムに跨って実装することが可能なため、ＪＤＷＰの使用により、マルチプラットフォームシステムを介した遠隔デバッグが容易となる。反対に、従来技術によるデバッグシステムの場合は、その多くが単一のプラットフォーム上で走るように設計されており、一般に、同一または類似のプラットフォームで走っているアプリケーションのみをデバッグすることができる。

【0003】

【発明が解決しようとする課題】通常は、デバッガアプリケーションがＪａｖａプログラミング言語で記述され、ターゲットサイドがネイティブコードで記述されている。ＪＤＷＰのリファレンス実装（reference implementation）では、フロントエンドのデバッガコンポーネントがＪａｖａで記述され、ターゲットＶＭのためのバックエンド・リファレンス実装がＣで記述されている。いずれのコードも詳細なプロトコル仕様に準拠している必要があり、そうでなければリファレンスシステムが不能となる。つまり、フロントエンドとバックエンドのコードが互いに互換性があり、且つプロトコル仕様に対して真に互換性があることを保証する、何らかのメカニズムが必要とされている。

【0004】言語は、例えばオブジェクトマネージメントグループ（ＯＭＧ：Object Management Group）が開発したＣＯＲＢＡ（Common Object Request Broker Architecture）の一部であるインターフェース定義言語（ＩＤＬ：Interface DefinitionLanguage）等のように、プロセス間／オブジェクト間通信の仕様を明記するために存在するものである。これらの言語はコンパイルされて（すなわちＩＤＬコンパイラによってコンパイルされて）、通信のクライアントサイドのためにスタブを、そしてサーバサイドのためにスケルトンをそれぞれ形成する。しかしながら、このような言語は、プロトコル準拠なデバッガコードの生成に伴う問題の解決を目指したものではない。

【0005】従って、デバッガのためのフロントエンドコードおよびバックエンドコードを、詳細な仕様から直接生成するための方法があることが望ましい。

【0006】

【課題を解決するための手段およびその作用・効果】本発明は、ＪＤＷＰ通信プロトコル等のインターフェース仕様に対して共に互換性を有するような、フロントエンドコードおよびバックエンドコードを自動的に生成するための方法を提供する。先ず、フロントエンドとバックエンドの間における通信プロトコルの記述を含んだ、詳細なプロトコル仕様を記述する。次いで、仕様の構文解析を行うコードジェネレータに、その詳細な仕様を入力する。すると、形式仕様から自動的にフロントエンドコードが生成されて、Ｊａｖａプログラミング言語等の第１のコンピュータ言語で書き込まれる。次いで、コードジェネレータによってバックエンドコードが生成されて、Ｃ等の第２のコンピュータ言語で書き込まれる。

【0007】本発明はさらに、人が判読可能なプロトコル仕様の記述を含んだ、ＨＴＭＬコードを生成しても良い。

【0008】

【発明の実施の形態】以下の説明は、当業者が本発明を実施および利用できるようにするために提供され、発明者によって考え出された本発明を実行するのに最良の形態を示すものである。しかしながら、本発明の基本原理は、第１の仮想マシン上を走るフロントエンドのデバッガプログラムと、第２の仮想マシン上を走るバックエンドのデバッガエージェントプログラムと、の間の通信プロトコルが形式仕様によって定義される場合に、フロントエンドプログラムとバックエンドプログラムの互換性を保証するための方法を特に提供するように定義されているため、当業者にとって、種々の変更態様の実施は依然として容易に明らかである。

【0009】本発明では、コンピュータシステムに格納されているデータを使用する、種々のコンピュータ実装オペレーションを利用する。これらのオペレーションは、物理量の物理的操作を必要とするものを含むが、それらに限定はされない。これらの物理量は、必然ではないものの、格納、転送、結合、比較、およびその他の操作が可能な電気または磁気信号の形を採るのが通常である。本発明の一部をなすこれらのオペレーションは、有用なマシンオペレーションである。実施される操作は、生成する、識別する、走らせる、決定する、比較する、

実行する、ダウンロードする、または検知する等の用語で呼ばれることが多い。これらの電気また磁気信号は、共同使用という基本的な理由から、ビット、値、要素、変数、文字、データ等の名称で呼ぶと便利なこともある。しかしながら、これらのおよび類似の用語が、全て適切な物理量に関連付けられて適用された便利なラベルに過ぎないことを、記憶に留めておく必要がある。

【００１０】本発明はまた、上述したオペレーションを実施するためのデバイス、システム、または装置に関する。このシステムは、必要な目的のために特別に構築されても良いし、あるいは、コンピュータに格納されたコンピュータプログラムにより選択的にアクティベイトまたは構成される汎用コンピュータであっても良い。上述したプロセスは、いかなる特定のコンピュータまたはその他の計算装置にも固有に関連するものではない。特に、ここで教示される内容に従って記述されたプログラムと共に、種々の汎用コンピュータを使用できる場合もあり、あるいは、必要なオペレーションの実施のために、より専門化されたコンピュータシステムを構築する方が便利な場合もある。

【００１１】図１は、本発明の一実施形態に従ってプロセスを実行するのに適した、汎用コンピュータシステム１００を示したブロック図である。図１には、汎用コンピュータシステムの一実施形態が示されているが、他のコンピュータシステムアーキテクチャおよび構成を使用して、本発明によるプロセスを実行することもできる。コンピュータシステム１００は、以下に挙げる種々のサブシステムから構成されており、少なくとも１つのマイクロプロセッササブシステム（中央処理装置またはＣＰＵとも呼ばれる）１０２を含む。つまりＣＰＵ１０２は、単チッププロセッサまたはマルチプルプロセッサによって実現されることができる。ここで、再構成が可能な計算機システムでは、ＣＰＵ１０２をプログラマブルな論理装置のグループ内で分散させられる点に注意が必要である。このようなシステムでは、プログラマブルな論理装置を、コンピュータシステム１００のオペレーション制御で必要とされるように再構成することができる。この方法を使用すると、入力データの操作がプログラマブルな論理装置のグループ内で分散される。ＣＰＵ１０２は、コンピュータシステム１００のオペレーションを制御する汎用デジタルプロセッサである。ＣＰＵ１０２は、メモリから引き出された命令を使用することにより、入力データの受信および操作、ならびに出力デバイス上へのデータの出力および表示を制御する。

【００１２】ＣＰＵ１０２は、メモリバス１０８を介して、通常はランダムアクセスメモリ（ＲＡＭ）である第１の一次記憶１０４に双方向的に接続され、通常は読取り専用メモリ（ＲＯＭ）である第２の一次記憶領域１０６に単方向的に接続されている。周知のように、一次記憶１０４は、汎用記憶領域としておよびスクラッチパッドメモリとして使用することができ、入力データおよび処理済みのデータを格納するのにも使用することができる。一次記憶１０４はまた、ＣＰＵ１０２上で作動するプロセスのための他のデータおよび命令に加えて、プログラミングの命令およびデータをデータオブジェクトの形で格納することもでき、通常は、メモリバス１０８上でデータおよび命令を双方向的に高速転送するのに使用される。同じく周知のように、一次記憶１０６は、基本オペレーティング命令、プログラムコード、ならびにＣＰＵ１０２がその機能を実施するのに使用するデータおよびオブジェクトを含むのが通常である。一次記憶装置１０４，１０６は、例えば、データアクセスの単方向性または双方向性に対する要求等に依存して、以下に挙げる任意の適切なコンピュータ読取り可能記憶媒体を含むことができる。ＣＰＵ１０２はまた、キャッシュメモリ１１０内の頻繁に必要とされるデータを、直接且つ非常に高速で引き出したり格納したりすることができる。

【００１３】取り外し可能な大容量記憶装置１１２は、コンピュータシステム１００に追加のデータ記憶容量を提供するものであり、周辺バス１１４を介してＣＰＵ１０２に単方向的または双方向的に接続されている。例えば、一般にＣＤ－ＲＯＭとして知られる取り外し可能な大容量記憶装置は、ＣＰＵ１０２に単方向的にデータを引き渡すのが通常であり、フロッピィディスクは、ＣＰＵ１０２に双方向的にデータを引き渡すことができる。記憶装置１１２はまた、磁気テープ、フラッシュメモリ、搬送波上に組み込まれた信号、ＰＣカード、ポータブルな大容量記憶装置、ホログラム記憶装置、およびその他の記憶装置等の、コンピュータ読取り可能媒体を含んでいて良い。固定大容量記憶１１６もまた、追加のデータ記憶容量を提供するものであり、周辺バス１１４を介してＣＰＵ１０２に双方向的に接続されている。大容量記憶１１６の最も代表的な例として、ハードディスクドライブが挙げられる。これらの媒体へのアクセスは、一次記憶１０４，１０６へのアクセスよりも低速であるのが通常である。大容量記憶１１２，１１６は、ＣＰＵ１０２によるアクティブな使用がなされていない追加のプログラミング命令、データ等を格納するのが通常である。ここで明らかなように、もし必要であれば、大容量記憶１１２，１１６内に保持されている情報を一次記憶１０４（例えばＲＡＭ）の一部分をなす仮想記憶として標準的な形で組み込んでも良い。

【００１４】周辺バス１１４は、ＣＰＵ１０２から記憶サブシステムへのアクセスを提供するのに加えて、その他のサブシステムおよびデバイスへのアクセスを提供するのにも使用される。上述した実施形態において、それらのサブシステムおよび装置は、表示モニタ１１８およびアダプタ１２０、プリンタ１２２、ネットワークインターフェース１２４、補助入出力デバイスインターフェース１２６、サウンドカード１２８およびスピーカ１３

0、ならびに必要に応じた他のサブシステムを含む。

【0015】ネットワークインターフェース124は、CPU102が、図示されたネットワーク接続を使用して別のコンピュータ、コンピュータネットワーク、またはテレコミュニケーションネットワークに接続できるようにする。CPU102は、上述したメソッド段階を実施する際に、ネットワークインターフェース124を介し、別のネットワークとの間で情報（例えばデータオブジェクトまたはプログラム命令）を送受信する場合があると考えられる。情報は、CPU上で実行される命令のシーケンスとして表されることが多く、例えば、搬送波に組み込まれたコンピュータデータ信号の形で別のネットワークとの間で送受信される場合もある。コンピュータシステム100を外部のネットワークに接続し、データを標準プロトコルにもとづいて転送するためには、インターフェースカードまたは類似のデバイス、およびCPU102により実行される適切なソフトウェアを使用することができる。すなわち、本発明による方法の実施形態は、CPU102上のみで実行されても良いし、インターネット、イントラネットネットワーク、またはローカルエリアネットワーク等のネットワークを通じ、処理を部分的に共用する遠隔CPUと関連して実施されても良い。また、追加の大容量記憶装置（未図示）が、ネットワークインターフェース124を介してCPU102に接続されていても良い。

【0016】補助入出力デバイスインターフェース126は、カスタマイズされた一般のインターフェースを表す。このインターフェースにより、CPU102は、マイクロフォン、タッチ検知ディスプレイ、トランスデューサカード読取り装置、テープ読取り装置、音声または筆跡認識装置、バイオメトリックス読取り装置、カメラ、ポータブルな大容量記憶装置、および他のコンピュータ等の他のデバイスにデータを送信することができ、より典型的にはこれらからデータを受信することができる。

【0017】CPU102には更に、ローカルバス134を介してキーボードコントローラ132が接続されている。キーボードコントローラ132は、キーボード136またはポインタ138からの入力を受信し、キーボード136またはポインタ138からのデコード後のシンボルをCPU102に送信するためのものである。ポインタは、マウス、スタイラス、トラックボール、またはタブレットで良く、グラフィカルユーザインターフェースとの相互作用に有用である。

【0018】本発明の実施形態はさらに、種々のコンピュータ実装オペレーションを実施するためのプログラムコードを含むコンピュータ読取り可能媒体を伴った、コンピュータ記憶製品に関する。コンピュータ読取り可能媒体は、データを格納し、格納したデータをコンピュータシステムに読み取らせることができる、任意のデータ記憶装置である。コンピュータ読取り可能媒体およびプログラムコードは、本発明の目的のために特別に設計および構築されたもので良く、あるいは、コンピュータソフトウェア技術の当業者に周知のものでも良い。コンピュータ読取り可能媒体は、上述した全ての媒体を含み、それらに限定されない。すなわち、ハードディスク、フロッピィディスク、磁気テープ等の磁気媒体、CD-ROMディスク等の光媒体、光フロッピィディスク等の光磁気媒体、ならびにアプリケーション固有の集積回路（ASIC）、プログラマブル論理装置（PLD）、ROMおよびRAMデバイス等の特別構成のハードウェアデバイス等である。コンピュータ読取り可能媒体が、結合コンピュータシステムのネットワーク上で搬送波に組み込まれたデータ信号として分散できるため、コンピュータ読取り可能コードは、分散方式によって格納および実行される。プログラムコードは、例えば、コンパイラ等により生成された機械コードと、インタプリタを使用して実行され得るハイレベルコードを含んだファイルとを、共に含んでいる。

【0019】当業者に明らかなように、上述したハードウェア要素およびソフトウェア要素は、標準的な設計および構成である。本発明への使用に適した別のコンピュータシステムは、追加のサブシステムを備えていて良く、あるいはより少ないサブシステムを備えていても良い。また、メモリバス108、周辺バス114、およびローカルバス134は、サブシステムの接続に供するあらゆる相互接続構造の例証となるものである。例えばローカルバスは、固定大容量記憶116およびディスプレイアダプタ120にCPUを接続するために使用することができる。図1に示されるコンピュータシステムは、本発明への使用に適したコンピュータシステムの一例に過ぎず、異なるサブシステム構成を有した他のコンピュータアーキテクチャも同様に利用することができる。

【0020】一実施形態において、本発明は、一般にJavaプラットフォームベースの分散アーキテクチャを実装したコンピュータシステムに適用可能であり、その視点から説明がなされている。しかしながら、以下の説明からわかるように、本発明の概念および方法論は、Javaプラットフォームベースの分散アーキテクチャに限定的だと解釈されるべきではない。このJavaプラットフォームベースのアーキテクチャは、好ましい実施形態を説明するためのみに使用されるものである。分散Javaプラットフォーム実装は、多くの異なるタイプのハードウェア、オペレーティングシステム、さらにはJava仮想マシン（VM）を有することができる。このため、全く異なるアーキテクチャの遠隔システム上を走っているプログラムを、デバッグする必要が生じる可能性もある。また多くの場合、「元の」状態になるべく近い状態でターゲットシステムをデバッグできるよう、実際には、これとは別のコンピュータシステム上にメイ

ンデバッガプログラムをロードすることが好ましい。図２ａに示されるように、Ｊａｖａプラットフォームデバッガアーキテクチャ（ＪＰＤＡ）は、３つの個々のインターフェースを定義することによりローカルデバッグおよびリモートデバッグをサポートしている。Ｊａｖａプラットフォームデバッガアーキテクチャは、デバッガアプリケーションの生成時に使用される一組のインターフェースを定義するものである。ＪＰＤＡは、Ｊａｖａデバッグインターフェース（ＪＤＩ）と、Ｊａｖａデバッグワイヤプロトコル（ＪＤＷＰ）と、Ｊａｖａ仮想マシンデバッグインターフェース（ＪＶＭＤＩ）とから構成される。ＪＰＤＡは、デバッガアプリケーションが直面している接続の問題全般を解決するものである。

【００２１】上述した実施形態において、第１のコンピュータシステム上では第１のＪａｖａ仮想マシン（ＶＭ）上をＪａｖａデバッガが走る。上述した実施形態での使用に適した本発明でのＪａｖａＶＭを、図４に示し、その内容は後述する。デバッガプログラムは、ハイレベルのＪａｖａデバッグインターフェース（ＪＤＩ）を実装するフロントエンドコンポーネント（以下では「フロントエンド」と称する）を有する。ユーザインターフェースを提供するデバッガアプリケーションは、ＪＤＩのクライアントである。デバッギ（debuggee）とは、デバッグされているプロセスであり、デバッグされているアプリケーションと、そのアプリケーションを走らせている第２のＪａｖａ仮想マシンと、「バックエンド」デバッガエージェント（以下では「バックエンド」と称する）とから構成される。バックエンドは、デバッガフロントエンドからデバッギ（ＶＭ）へリクエストを通信し、そのリクエストに対するレスポンスをフロントエンドへ送り返す役割を担っている。バックエンドは、Ｊａｖａデバッグワイヤプロトコル（ＪＤＷＰ）を使用した通信チャネルを介してフロンドエンドと通信し、Ｊａｖａ仮想マシンデバッグインターフェース（ＪＶＭＤＩ）を使用するデバッギＶＭと通信する。

【００２２】図２ｂは図２ａと類似しているが、本発明にトランスポートモジュールを追加するために必要となる２つの付加的コンポーネントが示されている。これら２つの追加の論理コンポーネントは、フロントエンドＪＤＷＰ処理モジュール（processing module）およびバックエンドＪＤＷＰ処理モジュールである。本発明によるデバッガプロトコルジェネレータは、フロントエンドおよびバックエンドのＪＤＷＰ処理モジュールを生成することを目標の１つとしている。図２ｂに示されている論理コンポーネントは、ベンダが固有の実装を組み込むことができる基本的なコンポーネントである。フロントエンドおよびバックエンドのトランスポートモジュールは、共用メモリ、ソケット、またはシリアルライン等のトランスポートメカニズムを実装するものである。

【００２３】このようにして、Ｊａｖａプラットフォームデバッガアーキテクチャは、図２ａおよび図２ｂに示されるように、別々に区別される３つのインターフェースをデバッグ用に提供する。第三者であるベンダは、自身のニーズに最も適合するインターフェースレベルを選択し、それに応じてデバッガアプリケーションを書く。具体的に言うと、ＪＤＩはフロントエンドに実装される１００％Ｊａｖａプラットフォームのインターフェースであり、ハイレベルで情報およびリクエストを定義するものである。ＪＤＰＡのグラフィカルユーザインターフェースにのみ集中したいベンダの場合は、このレベルを使用するだけで良い。

【００２４】ＪＶＭＤＩインターフェースは、デバッギＶＭによって実装されるネイティブコードインターフェースである。ＪＶＭＤＩインターフェースは、デバッグのためにＶＭが提供しなくてはならないサービスを定義し、情報、アクション、および通知へのリクエストを含む。デバッガのためにＶＭインターフェースを指定すると、ＪＰＤＡに任意のＶＭ実装をプラグインすることが可能となる。バックエンドは非ネイティブコードで書かれていても良いが、経験上、デバッギと同じＶＭサービスを共用するデバッガサポートコードを実行するとデッドロックその他の望ましくない動作を生じ得ることが知られている。

【００２５】ＪＤＷＰは、フロントエンドとバックエンドとの間で転送される情報およびリクエストのフォーマットを定義する。ＪＤＷＰは、フォーマット済みの情報を物理的に伝送するのに使用されるトランスポートメカニズムを指定していない。すなわち、プロセス間通信（ＩＰＣ）の形式は指定されていない。ソケット、シリアルライン、共用メモリ等の異なるトランスポートメカニズムを使用できる。通信プロトコルの仕様を明記することにより、デバッギおよびデバッガのフロントエンドを別々のＶＭ実装および／またはプラットフォーム上で走らせることが可能となる。また、中間インターフェースを定義することにより、フロントエンドをＪａｖａ言語以外の言語で、バックエンドを非ネイティブコード（すなわちＪａｖａ言語コード）でそれぞれ書くことが可能となる。分散インターフェースが使用されているため、ＪＶＭＤＩインターフェースに執着したくないＶＭベンダの場合はＪＤＷＰを介してアクセスを提供することもできる点に、注意が必要である。

【００２６】３つの個々のインターフェースを定義することにより、ＪａｖａプラットフォームデバッガアーキテクチャすなわちＪＰＤＡは、従来のデバッガシステムに関連した多くの限界を克服している。本発明は、インターフェースの文書化の問題、ならびにＪＤＷＰレベルにおける複数のプラットフォーム間およびプログラミング言語間での互換性管理の問題に対応するものである。ＪＤＩ層およびＪＶＭＤＩ層は従来のプログラミングインターフェースであるため、互換性および文書化の問題

がそれほど深刻ではなく、既存のツールに対してより適用しやすい。

【0027】互換性および文書化の問題は、互換性のコードおよび文書を単一の仕様源から生成することによって解決することができる。より厳密に言うと、この解決策は、フロントエンドJDWP処理モジュール（Java言語のフロントエンド実装の一部になる）と、バックエンドJDWP処理モジュール（C言語のバックエンド実装の一部になる）と、プロトコル仕様のHTML文書と、を生成することを含む。

【0028】生成されたフロントエンドJDWP処理モジュールによって実施されるタスクは、概して2つのカテゴリに配置することができる。1つ目のカテゴリは、デバッギVMで生成されたイベントに関する。このイベントは、バックエンドを介してフロントエンドに送信されなければならない。フロントエンドJDWP処理モジュールは、1）JDWPフォーマット化されたイベントをバックエンドから読み取って構文解析し、2）イベントをJDIイベントに変換し、3）イベントを待ち行列に入れる。もう1つのカテゴリは、JDIを介しデバッガアプリケーションによって作成されたリクエストに関する。フロントエンドJDWP処理モジュールは、1）JDWPフォーマット化されたリクエストをワイヤに書き込み、それらをバックエンドに送信し、2）適切な応答とそのリクエストとを関連付け、3）応答を読み取って構文解析し、4）応答をリクエスタに送信する。バックエンドJDWP処理モジュールは、通信のもう一端を処理しなければならないので、やはり2カテゴリの処理を有している。バックエンドJDWP処理モジュールは、イベント処理のためにワイヤに（JVMDIを介して生成された）イベントを書き込み、それをフロントエンドに送信する。バックエンドJDWP処理モジュールは、リクエスト処理のために、1）JDWPフォーマット化されたリクエストをフロントエンドから読み取って構文解析し、2）応答を生成する他のバックエンドコードにそのリクエストを転送し、3）応答をワイヤに書き込み、フロントエンドに送信する。

【0029】JDWP仕様、文書化、および実装コードの間の一貫性を機械的に確保することなしに、Javaプラットフォームデバッガアーキテクチャが、実働的なマルチベンダストラテジに発展することはなさそうである。このため本発明では、インターフェースの形式仕様を強制することによってその発展を補助する。関連技術は、この問題を解決できるように設計されていない。これらの関連技術においては、デバッガ実装コードを生成せずデバッガの問題点に対応した合理化がなされていないのである。

【0030】図3に示されるように、JDWPGenプログラムはJDWP（JDWP.spec）の形式仕様を構文解析し、その仕様から1）プロトコル文書（JDWPdetails.html）と、フロントエンドJDWP処理モジュール（JDWP.java）と、（現在のところ人の手によって書かれている）バックエンドJDWP処理モジュールの振る舞いを制御するC言語の「組み込み」ファイル（JDWPConstants.h）とを生成する。JDWP.javaおよびJDWPConstants.hが共に同じ仕様から生成されているため、デバッガコードの「デバッグ」が容易になり、2つの個々のプログラムを書き直すことなくJDWPの新ヴァージョンを容易に作成できるようになる。

【0031】本発明の一実施形態において、仕様言語は、JDWP仕様がJDWPGenによって正確に解釈できるように定義されている。この純粋な宣言言語はJDWP仕様言語である。以下これに関する説明を行う。JDWP仕様言語の構文は、括弧で囲んだステートメントでなっており、一般的な形式は：左括弧、ステートメントタイプ、引き数リスト、および右括弧となっている。これらのステートメントのネスティングは厳密には大きな制約を受けており、以下に挙げるJDWP仕様言語の文法によって正確に定義されている。

【0032】

13

```
SPECIFICATION
  NAME COMMENT SETLIST

SETLIST
  SET
  SETLIST SET

SET
   (CommandSet NAMEVALUE COMMANDLIST)
   (ConstantSet NAME CONSTANTLIST)

COMMANDLIST
  COMMAND
  COMMANDLIST COMMAND

COMMAND
  (Command NAMEVALUE COMMENT COMMANDBODY)

COMMANDBODY
  (Out STRUCTURE)(Reply STRUCTURE)
  (Event STRUCTURE)

STRUCTURE
  ELEMENT
  STRUCTURE ELEMENT


ELEMENT
  (DATATYPE NAME COMMENT)
  (Group NAME STRUCTURE)
  (Repeat NAME COMMENT ELEMENT)
  (Select NAME SELECTOR ALTLIST)

SELECTOR
  (INTEGRALDATATYPE NAME COMMENT)

ALTLIST
  ALT
  ALTLIST ALT

ALT
  (Alt NAMEVALUE COMMENT STRUCTURE)

DATATYPE
  INTEGRALDATATYPE
```

14

```
  boolean
  object
  threadObject
  threadGroupObject
  arrayObject
  stringObject
  classLoaderObject
  classObject
  referenceType
  referenceTypeID

  classType
  interfaceType
  arrayType
  method
  field
  frame
  string
  value
  location
  tagged-object
  referenceTypeID
  typed-sequence
  untagged-value

INTEGRALDATATYPE
  int
  long
  byte

CONSTANTLIST
  CONSTANT
  CONSTANTLIST CONSTANT

CONSTANT
  (Constant NAMEVALUE COMMENT)

NAMEVALUE
  NAME=NUMBER
  NAME=NAME
```

【００３３】全て大文字で記されている記号は非ターミナル（non-terminals）であり、その他の記号は全てターミナル(terminals)である。非ターミナルは、以下の場合を除いて文法内で定義される。

【００３４】NAME　　文字のシーケンス
NUMBER　　数字のシーケンス
COMMENT　二重引用符内のまたは何にも括られていない任意のテキスト

【００３５】仕様言語のセマンティックス

【００３６】リクエストコマンドは、フロントエンドで

作成された情報へのリクエストを指定している。ここで、Outセクションはリクエストを構成するデータのフォーマットを正確に指定し、Replyセクションはバックエンドによって返されるデータのフォーマットを正確に指定している。イベントコマンドは、バックエンドから発せられるイベント内のデータのフォーマットを正確に指定している。定数は、コマンド内で使用する特定の値を指定している。

【００３７】本実施例において、ＪＤＷＰＧｅｎは再帰的なディセントパーサを利用し、ＪＤＷＰ仕様言語で書かれているＪＤＷＰ仕様を構文解析する。生成ＬＡＬＲ（１）パーサ等の他の構文解析技術も、同様に使用することができる。パーサは、仕様を抽象構文ツリーで表現する。構文ツリーの各ノードは、そのノードの出力を生成するのに必要なアクションをカプセル化するオブジェクトである。ノードは、入力仕様内のステートメントに直接対応する。さらなる演算処理は、全てこの抽象構文ツリーを「ツリー探索」することによって達成される。多くのパスが、名前を解析してエラーをチェックするために使用される。最終的には、構文ツリーをさらに３回ツリー検索して出力が生成される。すなわち、ＪＤＷＰを介して情報の送受信を行うためにフロントエンドで使用されるＪａｖａクラスを生成するために１回、ＪＤＷＰを介して情報の送受信を行うためにバックエンドで使用される定義文を含んだＣ言語組み込みファイルを生成するために１回、人が読み取り可能なパブリッシュ化された仕様ドキュメントをＨＴＭＬで生成するために１回である。

【００３８】図４は、上述した図１のコンピュータシステム１００によってサポートされることができる、ＪＶＭ等の仮想マシンを描いた図である。ソースコード４０１は、コンパイル環境４０９内のバイトコードコンパイラ４０３に提供される。バイトコードコンパイラ４０３は、ソースコード４０１をバイトコード４０５に変換する。ソースコード４０１は通常、ソフトウェア開発者によって形成された時点でバイトコード４０５に変換される。

【００３９】バイトコード４０５は、複製およびダウンロードできるのが通常であり、あるいは、例えば図１のネットワークインターフェース１２４を介してネットワークで分散させたり、図１の一次記憶１０４等の記憶装置に格納したりすることができる。上述した実施形態において、バイトコード４０５はプラットフォーム独立である。すなわちバイトコード４０５は実質上、適切な仮想マシンを走らせている任意のコンピュータシステム上で実行することが可能である。バイトコードをコンパイルすることによって形成されるネイティブ命令は、ＪＶＭで後ほど使用するために保持される。この方法では、変換コストがかかる代わりに、多数の実行時には、インタープリタ化コードよりもネイティブコードが速度的に有利となる。例えばＪａｖａ（登録商標）環境では、ＪＶＭを走らせているコンピュータシステム上でバイトコード４０５を実行することができる。

【００４０】バイトコード４０５は、仮想マシン４１１を含むランタイム環境４１３に提供される。ランタイム環境４１３は、図１のＣＰＵ１０２等のプロセッサを使用して実行できるのが通常である。仮想マシン４１１は、コンパイラ４１５と、インタープリタ４１７と、ランタイムシステム４１９と、を含む。バイトコード４０５は一般に、コンパイラ４１５またはインタープリタ４１７のいずれかに提供することができる。

【００４１】バイトコード４０５がコンパイラ４１５に提供されると、バイトコード４０５に含まれるメソッドがネイティブマシン命令（図示されていない）にコンパイルされる。一方、バイトコード４０５がインタープリタ４１７に提供されると、バイトコード４０５は一度に１バイトコードづつインタープリタ４１７に読み込まれる。インタープリタ４１７は各バイトコードを読み込むごとにそのバイトコードによって定義されているオペレーションを実行する。通常、インタープリタ４１７は、バイトコード４０５の処理およびそれに関連付けられたオペレーションの実施を実質的に連続して行う。

【００４２】オペレーティングシステム４２１からメソッドを呼び出す際に、メソッドをインタープリタ化メソッドとして呼び出すことが決定されると、ランタイムシステム４１９はインタープリタ４１７からメソッドを獲得することができる。一方、メソッドをコンパイル化メソッドとして呼び出すことが決定されると、ランタイムシステム４１９はコンパイラ４１５をアクティベイトする。すると、コンパイラ４１５はバイトコード４０５からネイティブ・マシン命令を生成し、マシン語命令を実行する。マシン語命令は、一般に仮想マシン４１１が終了した時点で破棄される。仮想マシンのオペレーション、特にＪａｖａ（登録商標）仮想マシンは、ティム・リンドホルムおよびフランク・イェリンによるＪａｖａ（登録商標）仮想マシン仕様（ISBN 0-201-63452-X）において、より詳細に記述されている。よってこの仕様全体を、引用によって本明細書中に組み込むこととする。

【００４３】当業者には明らかなように、本発明の範囲および精神を離脱しない範囲において、上述した好ましい実施形態を、種々の適応例および変更態様で構成することが可能である。このため本発明は、添付した請求の範囲の範囲内において、以上に特記した以外の形態で実施することが可能である。

【図面の簡単な説明】

【図１】本発明を実装するのに適したコンピュータシステムを示したブロック図である。

【図２ａ】Ｊａｖａプラットフォームデバッガアーキテクチャを示した図である。

【図２ｂ】本発明によるＪＤＷＰ処理モジュールを示す

Ｊａｖａプラットフォームデバッガアーキテクチャを示した図である。
【図３】本発明によるデバッガプロトコルジェネレータの入力および出力を示した図である。
【図４】本発明による一実装での使用に適したＪａｖａ仮想マシンを示した図である。
【符号の説明】
１００…コンピュータシステム
１０２…ＣＰＵ
１０４…第１の一次記憶
１０６…第２の一次記憶
１０８…メモリバス
１１０…キャッシュメモリ
１１２…取り外し可能な大容量記憶
１１４…周辺バス
１１６…固定大容量記憶
１１８…表示モニタ
１２０…ディスプレイアダプタ
１２２…プリンタ
１２４…ネットワークインターフェース
１２６…補助入出力デバイスインターフェース
１２８…サウンドカード
１３０…スピーカ
１３２…キーボードコントローラ
１３４…ローカルバス
１３６…キーボード
１３８…ポインタ
４０１…ソースコード
４０３…バイトコードコンパイラ
４０５…バイトコード
４０９…コンパイル環境
４１１…仮想マシン
４１３…ランタイム環境
４１５…コンパイラ
４１７…インタープリタ
４１９…ランタイムシステム
４１２…オペレーティングシステム

【図２ａ】

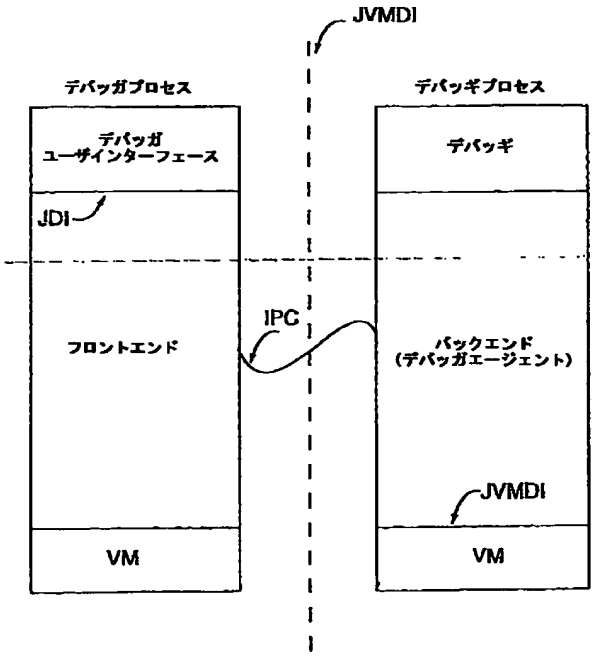

Figure 2a

【図２ｂ】

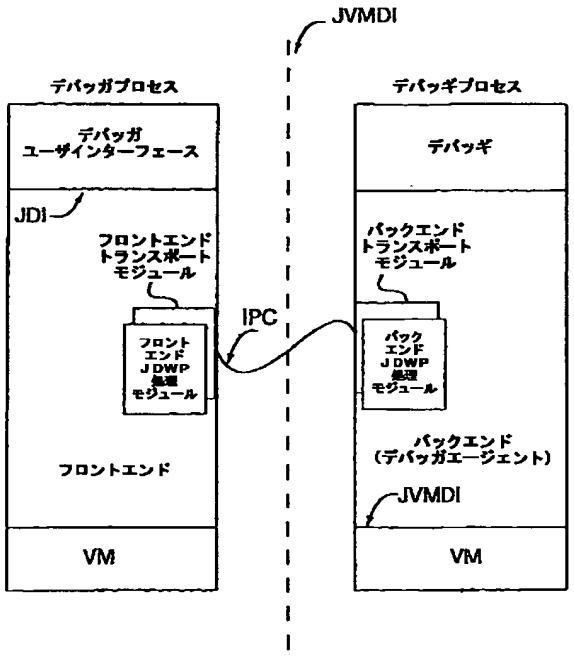

Figure 2b

【図1】

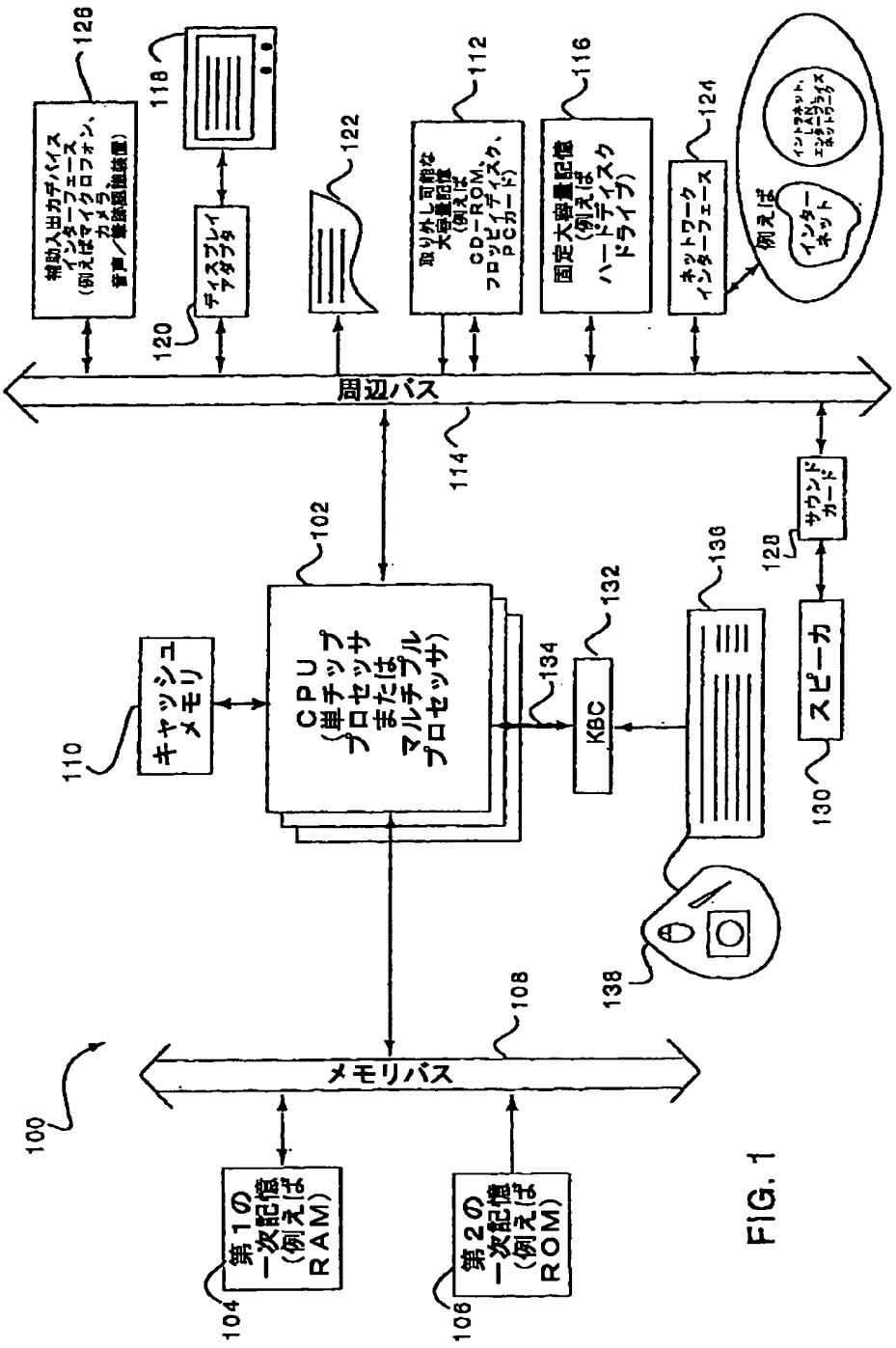
126
118
122
112
116
124
120
周辺バス
114
102
108
メモリバス
100
104
106
110
キャッシュメモリ
CPU
（単チッププロセッサまたはマルチプルプロセッサ）
134
KBC
132
136
138
サウンドカード
128
スピーカ
130
例えば
インターネット

FIG. 1

【図3】

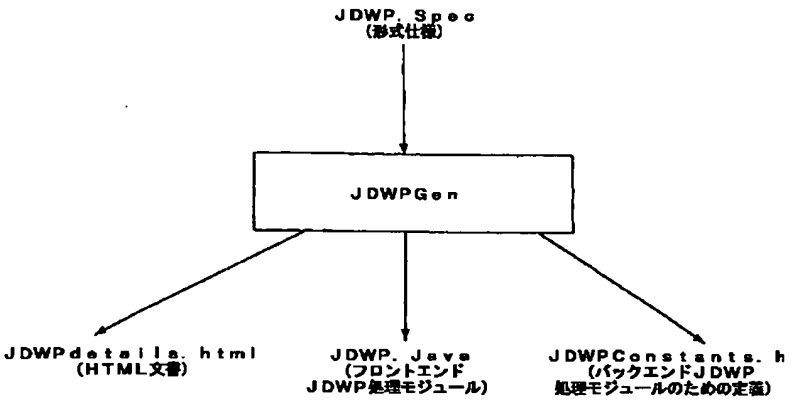

FIG. 3

【図4】

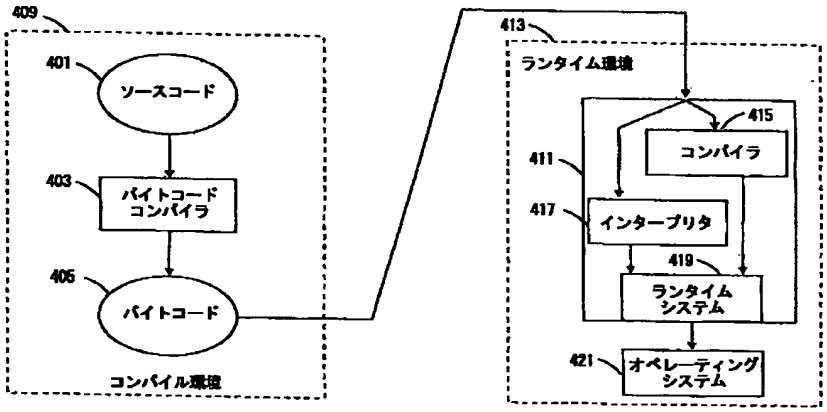

FIG. 4

フロントページの続き

(71)出願人　591064003
901 SAN ANTONIO ROAD
PALO ALTO, CA 94303, U. S. A.

(72)発明者　ロバート・ジー．・フィールド
アメリカ合衆国　カリフォルニア州95065
サンタ・クルーズ，ノース・ピッピン・レイン，270

(72)発明者　ゴードン・ハーシュ
アメリカ合衆国　カリフォルニア州94087
サニーベイル，ディー・ベルヴィル・ウェイ，1663

【外国語明細書】

1. Title of Invention

DEBUGGER PROTOCOL GENERATOR

2. Claims

1. A method for assuring compatibility between a formal specification, a front-end debugger program, and a back-end debugger program which interfaces with a debuggee system, the method comprising:

inputting a formal specification into a code generator;

utilizing the code generator to parse the formal specification;

generating a front-end debugger program portion from the formal specification; and

generating a back-end debugger program portion from the formal specification.

2. The method of Claim 1, wherein the front-end debugger program runs on a first virtual machine.

3. The method of Claim 2, wherein the front-end debugger program portion comprises Java programming language code.

4. The method of Claim 1, wherein the back-end debugger program directly controls and communicates with a second virtual machine.

5. The method of Claim 4, wherein the back-end debugger program portion comprises C language code.

6. The method of Claim 1, wherein the formal specifications is a Java Debug Wire Protocol specification.

7. The method of Claim 6, further comprising generating HTML code that contains a human-readable description of the protocol specification.

8. A method of Claim 1 further comprising enabling a communication protocol between the front-end program and the back-end program, wherein the communication protocol is defined by the formal specification.

9. The method of Claim 1, wherein the formal specification is written in a declarative language.

10. The method of Claim 8, wherein the communication protocol is a Java Debug Wire Protocol.

11. The method of Claim 8, further comprising generating HTML code from the formal specification that contains a human readable description of the communication protocol defined by the formal specification.

12. A method for automatically generating front-end debugger interface code and back-end debugger agent interface code that are both compatible with a communication protocol, the method comprising:

writing a formal specifications file that contains a description of a communication protocol between the front-end debugger code and the back-end debugger agent code:

inputting the formal specification file into a code generator:

utilizing the code generator to parse the formal specification:

generating the front-end debugger interface code from the formal specification; and

generating the back-end debugger agent interface code from the formal specification.

13. The method of Claim 12, wherein the front-end debugger interface code comprises Java code, the back-end debugger agent interface code comprises C code, and the formal specification comprises a specification language.

14. The method of Claim 13, wherein the communication protocol is a Java Debug Wire Protocol.

15. A computer readable medium including computer program code for automatically generating front-end debugger interface code and back-end debugger interface code that are both compatible with a communication protocol, the computer readable medium comprising:

computer program code for inputting a formal protocol specification into a code generator;

computer program code for utilizing the code generator to parse the formal protocol specification;

computer code for generating front-end debugger interface computer code from the specification; and

computer code for generating back-end debugger interface computer code from the specification.

16. The medium of claim 15, further comprising computer code for generating HTML code containing a human-readable description of the communication protocol.

17. The medium of Claim 16, wherein the communication protocol is a Java Debug Wire Protocol.

18. A computer system for automatically generating front-end debugger interface code and back-end debugger interface code that are both compatible with an communication protocol, the computer system comprising:

a processor; and

a computer program operating on the processor that reads in a formal communication protocol specification, parses the specification, and generates front-end debugger interface code and back end debugger interface code, such that the front-end code and the back-end code are fully compliant with the specification and compatible with each other.

3. Detailed Description of Invention

The present invention relates generally to the field of computer software, and more particularly to protocol generating software for generating software components from a formal specification.

The Java(trademark) Debug Wire Protocol (JDWP) (Java(trademark) and related marks are trademarks of Sun Microsystems, Inc.) is a protocol for communicating between a debugger application and a Java Virtual Machine (target VM). By implementing the JDWP, a debugger can either work in a different process on the same computer, or on a remote computer. Since Java(trademark) programming applications may be implemented across a wide variety of different hardware platforms and operating systems, the JDWP facilitates remote debugging across a multi-platform system. In contrast, many prior art debugging systems are designed to run on a single platform and most generally debug only applications running on the same or similar platform.

Typically, a debugger application is written in the Java programming language and the target side is written in native code. In a reference implementation of JDWP, a front-end debugger component is written in Java

and a back-end reference implementation for the target VM is written in C. Both pieces of code need to be compliant with a detailed protocol specification, or the reference system will fail. What is needed is some mechanism to assure that both the front-end and back-end code portions are truly compatible with the protocol specification and with each other.

Languages exist for the specification of inter-process/object communication, such as the Interface Definition Language (IDL) which is part of the Common Object Request Broker Architecture (CORBA), developed by the Object Management Group (OMG). These languages are compiled (i.e. by an IDL compiler) to produce stubs for the client side of communication and skeletons for the server side. However, such languages are not directed to the problems associated with generating protocol compliant debugger code.

Therefore, it would be desirable to have method for generating both the front-end code and the back-end code for a debugger directly from a detailed specification.

The present invention provides a method for automatically generating front-end code and back-end code that are both compatible with an interface specification, such as the JDWP communications protocol. First, a detailed protocol specification is written that contains a description of a communications protocol between the front-end and the back-end. The detailed specification is then input into a code generator that parses the specification. The front-end code is then automatically generated from the formal specification, and may be written in a first computer language such as the Java programming language. The code generator then generates the back-end code, which may be written in a second computer language such as C.

The present invention may further generate HTML code containing a human-readable description of the protocol specification.

The following description is provided to enable any person skilled in the art to make and use the invention and sets forth the best modes contemplated by the inventor for carrying out the invention. Various modifications, however, will remain readily apparent to those skilled in the art, since the basic principles of the present invention have been defined herein specifically to provide a method for assuring compatibility between a front-end debugger program running on a first virtual machine and a back-end debugger agent program running on a second virtual machine, wherein a communications protocol between the front-end program and the back-end program is defined by a formal specification.

The present invention employs various computer-implemented operations involving data stored in computer systems. These operations include, but are not limited to, those requiring physical manipulation of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. The operations described herein that form part of the invention are useful machine operations. The manipulations performed are often referred to in terms, such as, producing, identifying, running, determining, comparing, executing, downloading, or detecting. It is sometimes convenient, principally for reasons of common usage, to refer to these electrical or magnetic signals as bits, values, elements, variables, characters, data, or the like. It should remembered, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

The present invention also relates to a device, system or apparatus for

r performing the aforementioned operations. The system may be specially constructed for the required purposes, or it may be a general purpose computer selectively activated or configured by a computer program stored in the computer. The processes presented above are not inherently related to any particular computer or other computing apparatus. In particular, various general-purpose computers may be used with programs written in accordance with the teachings herein, or, alternatively, it may be more convenient to construct a more specialized computer system to perform the required operations.

FIG. 1 is a block diagram of a general purpose computer system 100 suitable for carrying out the processing in accordance with one embodiment of the present invention. Figure 1 illustrates one embodiment of a general purpose computer system. Other computer system architectures and configurations can be used for carrying out the processing of the present invention. Computer system 100, made up of various subsystems described below, includes at least one microprocessor subsystem (also referred to as a central processing unit, or CPU) 102. That is, CPU 102 can be implemented by a single-chip processor or by multiple processors. It should be noted that in re-configurable computing systems, CPU 102 can be distributed amongst a group of programmable logic devices. In such a system, the programmable logic devices can be reconfigured as needed to control the operation of computer system 100. In this way, the manipulation of input data is distributed amongst the group of programmable logic devices. CPU 102 is a general purpose digital processor which controls the operation of the computer system 100. Using instructions retrieved from memory, the CPU 102 controls the reception and manipulation of input data, and the output and display of data on output devices.

CPU 102 is coupled bi-directionally with a first primary storage 104.

typically a random access memory (RAM), and uni-directionally with a sec ond primary storage area 106, typically a read-only memory (ROM), via a memory bus 108. As is well known in the art, primary storage 104 can be used as a general storage area and as scratch-pad memory, and can also be used to store input data and processed data. It can also store progr amming instructions and data, in the form of data objects, in addition t o other data and instructions for processes operating on CPU 102, and is used typically used for fast transfer of data and instructions in a bi-directional manner over the memory bus 108. Also as well known in the a rt, primary storage 106 typically includes basic operating instructions, program code, data and objects used by the CPU 102 to perform its funct ions. Primary storage devices 104 and 106 may include any suitable comp uter-readable storage media, described below, depending on whether, for example, data access needs to be bi-directional or uni-directional. CPU 102 can also directly and very rapidly retrieve and store frequently ne eded data in a cache memory 110.

A removable mass storage device 112 provides additional data storage c apacity for the computer system 106, and is coupled either bi-directiona lly or uni-directionally to CPU 102 via a peripheral bus 114. For examp le, a specific removable mass storage device commonly known as a CD-ROM typically passes data uni-directionally to the CPU 102, whereas a floppy disk can pass data bi-directionally to the CPU 102. Storage 112 may al so include computer-readable media such as magnetic tape, flash memory, signals embodied on a carrier wave, PC-CARDS, portable mass storage devi ces, holographic storage devices, and other storage devices. A fixed mas s storage 116 also provides additional data storage capacity and is coup led bi-directionally to CPU 102 via peripheral bus 114. The most common example of mass storage 116 is a hard disk drive. Generally, access to t hese media is slower than access to primary storages 104 and 106.

Mass storage 112 and 116 generally store additional programming instructions, data, and the like that typically are not in active use by the CPU 102. It will be appreciated that the information retained within mass storage 112 and 116 may be incorporated, if needed, in standard fashion as part of primary storage 104 (e.g. RAM) as virtual memory.

In addition to providing CPU 102 access to storage subsystems, the peripheral bus 114 is used to provide access other subsystems and devices as well. In the described embodiment, these include a display monitor 118 and adapter 120, a printer device 122, a network interface 124, an auxiliary input/output device interface 126, a sound card 128 and speakers 130, and other subsystems as needed.

The network interface 124 allows CPU 102 to be coupled to another computer, computer network, or telecommunications network using a network connection as shown. Through the network interface 124, it is contemplated that the CPU 102 might receive information, e.g., data objects or program instructions, from another network, or might output information to another network in the course of performing the above-described method steps. Information, often represented as a sequence of instructions to be executed on a CPU, may be received from and outputted to another network, for example, in the form of a computer data signal embodied in a carrier wave. An interface card or similar device and appropriate software implemented by CPU 102 can be used to connect the computer system 100 to an external network and transfer data according to standard protocols.

That is, method embodiments of the present invention may execute solely upon CPU 102, or may be performed across a network such as the Internet, intranet networks, or local area networks, in conjunction with a remote CPU that shares a portion of the processing. Additional mass storage devices (not shown) may also be connected to CPU 102 through network interface 124.

Auxiliary I/O device interface 126 represents general and customized i nterfaces that allow the CPU 102 to send and, more typically, receive da ta from other devices such as microphones, touch-sensitive displays, tra nsducer card readers, tape readers, voice or handwriting recognizers, bi ometrics readers, cameras, portable mass storage devices, and other comp uters.

Also coupled to the CPU 102 is a keyboard controller 132 via a local b us 134 for receiving input from a keyboard 136 or a pointer device 138, and sending decoded symbols from the keyboard 136 or pointer device 138 to the CPU 102. The pointer device may be a mouse, stylus, track ball, or tablet, and is useful for interacting with a graphical user interface .

In addition, embodiments of the present invention further relate to co mputer storage products with a computer readable medium that contain pro gram code for performing various computer-implemented operations. The c omputer-readable medium is any data storage device that can store data w hich can thereafter be read by a computer system. The media and program code may be those specially designed and constructed for the purposes o f the present invention, or they may be of the kind well known to those of ordinary skill in the computer software arts. Examples of computer-r eadable media include, but are not limited to, all the media mentioned a bove: magnetic media such as hard disks, floppy disks, and magnetic tap e; optical media such as CD-ROM disks; magneto-optical media such as flo ptical disks; and specially configured hardware devices such as applicat ion-specific integrated circuits (ASICs), programmable logic devices (PL Ds), and ROM and RAM devices. The computer-readable medium can also be distributed as a data signal embodied in a carrier wave over a network o f coupled computer systems so that the computer-readable code is stored and executed in a distributed fashion. Examples of program code include

both machine code, as produced, for example, by a compiler, or files containing higher level code that may be executed using an interpreter.

It will be appreciated by those skilled in the art that the above described hardware and software elements are of standard design and construction. Other computer systems suitable for use with the invention may include additional or fewer subsystems. In addition, memory bus 108, peripheral bus 114, and local bus 134 are illustrative of any interconnection scheme serving to link the subsystems. For example, a local bus could be used to connect the CPU to fixed mass storage 116 and display adapter 120. The computer system shown in FIG. 1 is but an example of a computer system suitable for use with the invention. Other computer architectures having different configurations of subsystems may also be utilized.

In a described embodiment of the present invention, the invention is generally applicable to and described in terms of computer systems implementing a Java Platform based distributed architecture. However, as will be seen in the following description, the concepts and methodologies of the present invention should not be construed to be limited to a Java Platform based distributed architecture. Such an architecture is used only to describe a preferred embodiment. A distributed Java platform implementation may have many different types of hardware, operating systems, and even Java Virtual Machines (VMs). Therefore it may be necessary to debug a program running on a remote system, having completely different architecture. Also, in many instances it is preferable to actually load a main debugger program on a separate computer system so that the target system can be debugged in a state as close to possible to its "original" state. As shown in Figure 2a, the Java Platform Debugger Architecture (JPDA) supports local and remote debugging by defining three separate interfaces. The Java Platform Debugger Architecture defines a set of interfaces used in the creation of debugger applications. It consists of

the Java Debug Interface (JDI) and the Java Debug Wire Protocol (JDWP), and the Java Virtual Machine Debug Interface (JVMDI). The JPDA provides a solution to the general connection problems encountered by debugger applications.

In the described embodiment, on a first computer system, a Java debugger runs on a first Java Virtual Machine (VM). A Java VM suitable for use in the described embodiment of the present invention is shown and described in Figure 4 below. The debugger program has a front-end component (hereinafter "front-end") that implements a high-level Java Debug Interface (JDI). A debugger application, which provides a user interface, is a client of the JDI. The debuggee is the process that is being debugged, and it consists of the application being debugged, a second Java Virtual Machine running the application, and a "back-end" debugger agent (hereinafter "back-end"). The back-end is responsible for communicating requests from the debugger front-end to the debuggee (VM) and for communicating the response to the requests back to the front-end. The back-end communicates with the front-end over a communications channel using the Java Debug Wire Protocol (JDWP). The back-end communicates with the debuggee VM using the Java Virtual Machine Debug Interface (JVMDI).

Figure 2b is similar to Figure 2a but shows two additional components needed to enable the present invention plus transport modules. The two additional logical components are a front-end JDWP processing module and a back-end JDWP processing module. One of the goals of the debugger protocol generator of the present invention is to generate front-end and back-end JDWP processing modules. The logical components shown in Figure 2b are basic components for which vendors can provide their own implementations. The front-end and back-end transport modules implement a transport mechanism, such as shared memory, socket, or serial line.

Thus, as is shown in Figures 2a and 2b, the Java Platform Debugger Arc

hitecture provides three separate and distinct interfaces for debugging. Third-party vendors can choose which interface level best suits their needs and write a debugger application accordingly. Specifically, the JDI is a 100% Java platform interface implemented by the front-end, which defines information and requests at a high level. For vendors who wish to concentrate on a graphical user interface for the JDPA, they only need to use this level.

The JVMDI interface is a native code interface implemented by the debuggee VM. It defines the services that a VM must provide for debugging and includes requests for information, actions, and notifications. Specifying the VM interface for a debugger allows any VM implementation to plug into the JPDA. The back-end may be written in non-native code, but experience has shown that debugger support code running sharing the same VM services as the debuggee can cause deadlocks and other undesired behavior.

The JDWP defines the format of information and requests transferred between the front-end and the back-end. It does not specify the transport mechanism used to physically transmit the formatted information, that is, the form of inter-process communication (IPC) is not specified. Different transport mechanisms may be used such as sockets, serial line, shared memory, etc. The specification of the communication protocol allows the debuggee and the debugger front-end to run under separate VM implementations and/or on separate platforms. Also, by defining an intermediate interface, the front-end may be written in a language other than the Java language, or the back-end in non-native code (i.e. Java language code). Note that due to the use of distributed interfaces, a VM vendor that does not wish to adhere to the JVMDI interface can still provide access via the JDWP.

By defining three separate interfaces, the Java Platform Debugger Arch

itecture, JPDA overcomes many limitations associated with prior art debu gger systems. The present invention addresses the problem of documentin g the interface and of managing compatibility across multiple platforms and programming languages at the JDWP level. Because the JDI and JVMDI layers are conventional programming interfaces, the compatibility and do cumentation problems are less severe and more amenable to existing tools .

The compatibility and documentation problems are solved by generating compatible code and documentation from a single specification source. M ore specifically, this involves generating a front-end JDWP processing m odule (which becomes part of the front-end implementation in the Java la nguage), a back-end JDWP processing module (which will become part of th e back-end implementation in the C language), and HTML documentation of the protocol specification.

The tasks performed by the generated front-end JDWP processing module can be placed generally in two categories. One category relates to event is generated in the debuggee VM, which must be sent to the front-end thr ough the back-end. The front-end JDWP processing module: 1) reads and p arses JDWP formatted events from the back-end; 2) converts the events in to JDI events; and 3) queues the events. The other category relates to requests made through the JDI by the debugger application. The front-en d JDWP processing module: 1) writes JDWP formatted requests to the wire. sending them to the back-end; 2) associates the appropriate reply to th e request; 3) reads and parses the reply; and 4) delivers the reply to t he requestor. The back-end JDWP processing module must handle the other end of the communication, so it too has two categories of processing. For event processing, the back-end JDWP processing module writes the eve nt (which was generated through the JVMDI) to the wire, sending it to th e front-end. For requesting processing, the back-end JDWP processing mo

dule: 1) reads and parses JDWP formatted requests from the front-end; 2) forwards the request to other back-end code, which will generate a reply; 3) writes the reply to the wire, sending it to the front-end.

Without a mechanically assured consistency between the JDWP specification, documentation and implementation code, it is unlikely that the Java Platform Debugger Architecture could evolve into a workable multi-vendor strategy. Thus, the present invention enforces a formal specification of the interface and thereby aids its evolution. The related art has not been designed to solve this problem in that it does not generate debugger implementation code and is not streamlined to the problems of debuggers.

As shown in Figure 3, a JDWPGen program parses a formal specification of the JDWP (JDWP.spec), and from the specification generates: 1) the protocol documentation (JDWPdetails.html), the front-end JDWP processing module (JDWP.java), and a C language "include" file (JDWPConstants.h) which controls the behavior of the back-end JDWP processing module (which is presently manually written). Since both the JDWP.java and JDWPConstants.h are generated from the same specification, it is much easier to "debug" the debugger code, and to produce new versions of the JDWP without having to re-write two separate programs.

In one embodiment of the present invention, a specification language is defined so that the JDWP specification can be precisely interpreted by JDWPGen. This purely declarative language is the JDWP specification language, and is described below. The syntax of the JDWP specification language primarily consists of parenthesized statements with the general form: open parenthesis, statement type, argument list and close parenthesis. The argument list often consists of statements. The exact nesting these statements may have is highly constrained and is defined precisely by the following grammar for the JDWP specification language:

```
SPECIFICATION
        NAME COMMENT SETLIST
SETLIST
        SET
        SETLIST SET
SET
        ( CommandSet NAMEVALUE COMMANDLIST )
        ( ConstantSet NAME CONSTANTLIST )
COMMANDLIST
        COMMAND
        COMMANDLIST COMMAND
COMMAND
        ( Command NAMEVALUE COMMENT COMMANDBODY )
COMMANDBODY
        ( Out STRUCTURE ) ( Reply STRUCTURE )
        ( Event STRUCTURE )
STRUCTURE
        ELEMENT
        STRUCTURE ELEMENT
ELEMENT
        ( DATATYPE NAME COMMENT )
```

(Group-NAME STRUCTURE)

(Repeat NAME COMMENT ELEMENT)

(Select NAME SELECTOR ALTLIST)

SELECTOR

(INTEGRALDATATYPE NAME COMMENT)

ALTLIST

ALT

ALTLIST ALT

ALT

(Alt NAMEVALUE COMMENT STRUCTURE)

DATATYPE

INTEGRALDATATYPE

boolean

object

threadObject

threadGroupObject

arrayObject

stringObject

classLoaderObject

classObject

referenceType

referenceTypeID

classType

interfaceType

arrayType

method

field

frame

string

value

location

tagged-object

referenceTypeID

typed-sequence

untagged-value

INTEGRALDATATYPE

int

long

byte

CONSTANTLIST

CONSTANT

CONSTANTLIST CONSTANT

CONSTANT

(Constant NAMEVALUE COMMENT)

NAMEVALUE

NAME = NUMBER

NAME = NAME

The symbols in all capital letters are non-terminals and all other symbols are terminals. Non-terminals are defined within the grammar except f

or the following:

NAME a sequence of letters

NUMBER a sequence of digits

COMMENT arbitrary text within double quotes or nothing

Semantics of Specification Language

A request command specifies a request for information made by the front-end where the Out section exactly specifies the format of the data that makes up the request and the Reply section exactly specifies the format of the data that will be returned by the back-end. An event command exactly specifies the format of data in an event emanating from the back-end. Constants specify specific values for use within commands.

In the present embodiment, JDWPGen employs a recursive descent parser to parse the JDWP specification, which is written in the JDWP specification language. Other parsing techniques could be used as well, such as a generated LALR(1) parser. The parser constructs an abstract syntax tree representation of the specification. Each node in the tree is an object that encapsulates the actions needed to generate the outputs for that node. The nodes correspond directly with statements in the input specification. All further processing is accomplished by "walking" this abstract syntax tree. Several passes are used to resolve names and check for errors. Finally, the tree is walked three more times to generate the outputs: once to generate the Java class which is used by the front-end to send and receive information across JDWP; once to generate the C include file containing the definitions used by the back-end to send and receive information across JDWP; and once to generate the published human-readable specification document in HTML.

Figure 4 is a diagrammatic representation of a virtual machine, such as a JVM, that can be supported by computer system 100 of Figure 1 described above. Source code 401 is provided to a bytecode compiler 403 within a compile-time environment 409. Bytecode compiler 403 translates source code 401 into bytecodes 405. In general, source code 401 is translated into bytecodes 405 at the time source code 401 is created by a software developer.

Bytecodes 405 can generally be reproduced, downloaded, or otherwise distributed through a network, e.g., through network interface 124 of Figure 1, or stored on a storage device such as primary storage 104 of Figure 1. In the described embodiment, bytecodes 405 are platform independent. That is, bytecodes 405 may be executed on substantially any computer system that is running a suitable virtual machine. Native instructions formed by compiling bytecodes may be retained for later use by the JVM. In this way the cost of the translation are amortized over multiple executions to provide a speed advantage for native code over interpreted code. By way of example, in a Java(trademark) environment, bytecodes 405 can be executed on a computer system that is running a JVM.

Bytecodes 405 are provided to a runtime environment 413 which includes a virtual machine 411. Runtime environment 413 can generally be executed using a processor such as CPU 102 of Figure 1. Virtual machine 411 includes a compiler 415, an interpreter 417, and a runtime system 419. Bytecodes 405 can generally be provided either to compiler 415 or interpreter 417.

When bytecodes 405 are provided to compiler 415, methods contained in bytecodes 405 are compiled into native machine instructions (not shown). On the other hand, when bytecodes 405 are provided to interpreter 417, bytecodes 405 are read into interpreter 417 one bytecode at a time. Interpreter 417 then performs the operation defined by each bytecode as ea

ch bytecode is read into interpreter 417. In general, interpreter 417 p rocesses bytecodes 405 and performs operations associated with bytecodes 405 substantially continuously.

When a method is called from an operating system 421, if it is determi ned that the method is to be invoked as an interpreted method, runtime s ystem 419 can obtain the method from interpreter 417. If, on the other hand, it is determined that the method is to be invoked as a compiled me thod, runtime system 419 activates compiler 415. Compiler 415 then gene rates native machine instructions from bytecodes 405, and executes the m achine-language instructions. In general, the machine-language instruct ions are discarded when virtual machine 411 terminates. The operation o f virtual machines or, more particularly, Java(trademark) virtual machin es, is described in more detail in The Java(trademark) Virtual Machine S pecification by Tim Lindholm and Frank Yellin (ISBN 0 201-63452-X), whic h is incorporated herein by reference in its entirety.

Those skilled in the art will appreciate that various adaptations and modifications of the just-described preferred embodiments can be configu red without departing from the scope and spirit of the invention. Theref ore, it is to be understood that, within the scope of the appended claim s, the invention may be practiced other than as specifically described h erein.

4. Brief Description of Drawings

Figure 1 is a block diagram of a computer system suitable for implemen ting the present invention;

Figure 2a is a diagram illustrating the Java Platform Debugger Archite cture;

Figure 2b is a diagram illustrating the Java Platform Debugger Archite

cture showing JDWP processing modules of the present invention;

Figure 3 is a diagram illustrating the input and outputs of the debugger protocol generater of the present invention; and

Figure 4 is diagram of a Java Virtual Machine suitable for use in one implementation of the present invention.

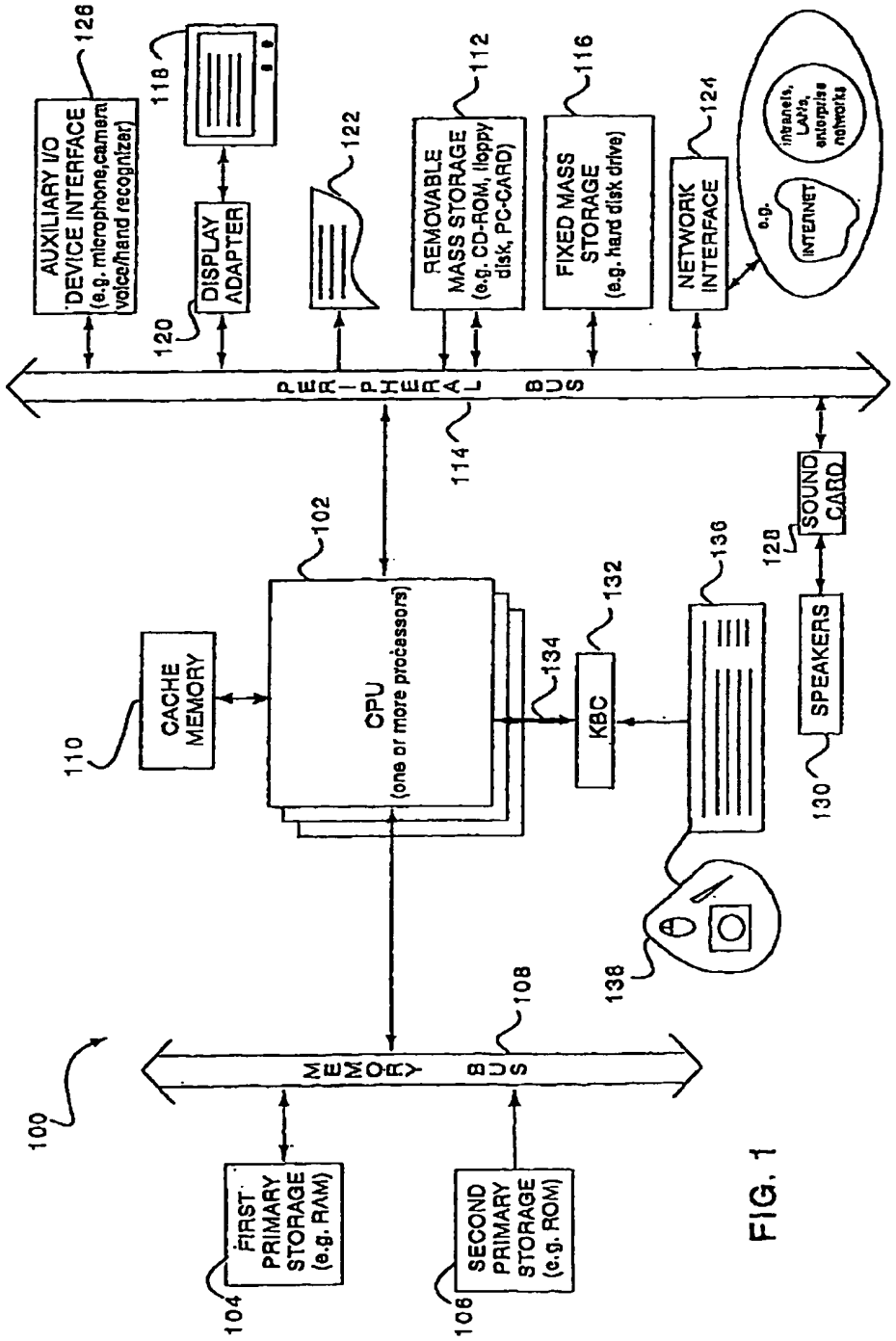

FIG. 1

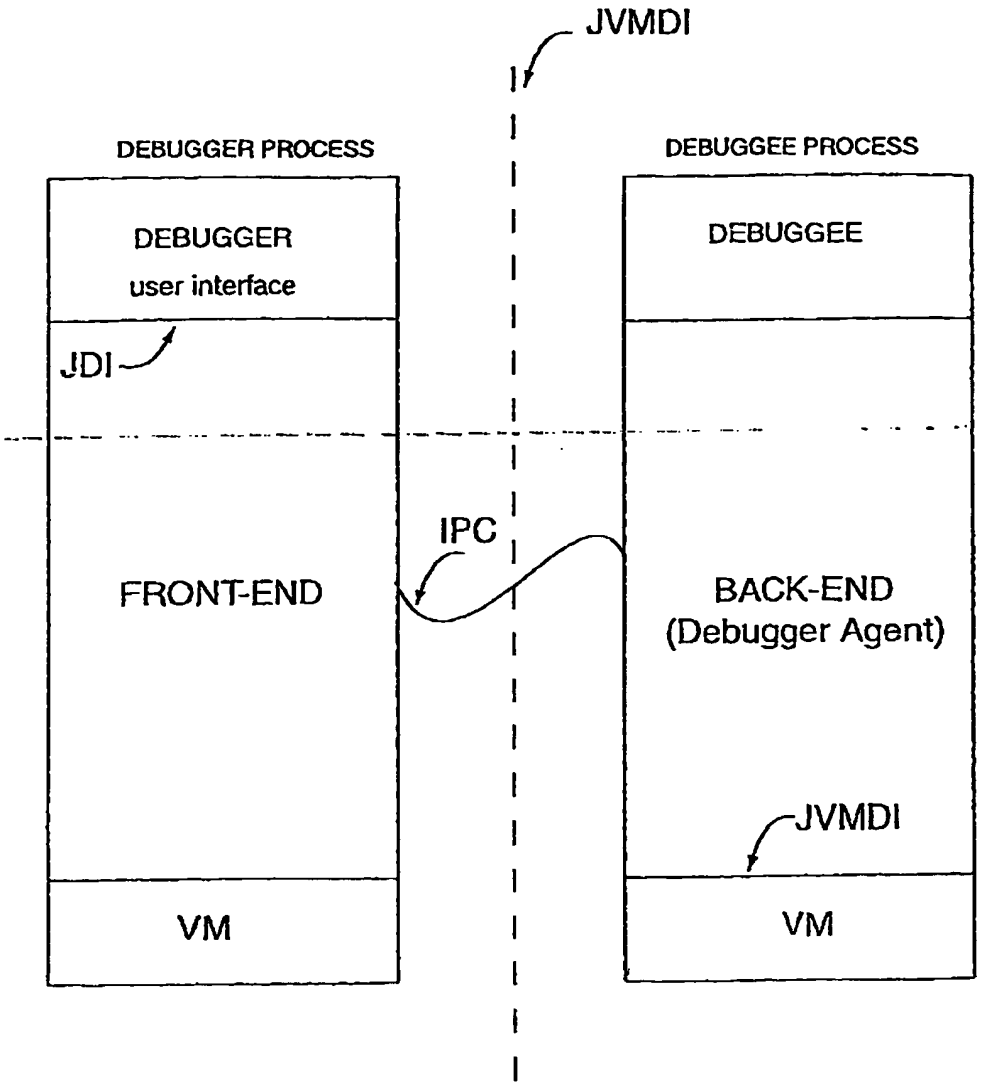
JVMDI
DEBUGGER PROCESS
DEBUGGEE PROCESS
DEBUGGER
user interface
DEBUGGEE
JDI
IPC
FRONT-END
BACK-END
(Debugger Agent)
JVMDI
VM
VM

Figure 2a

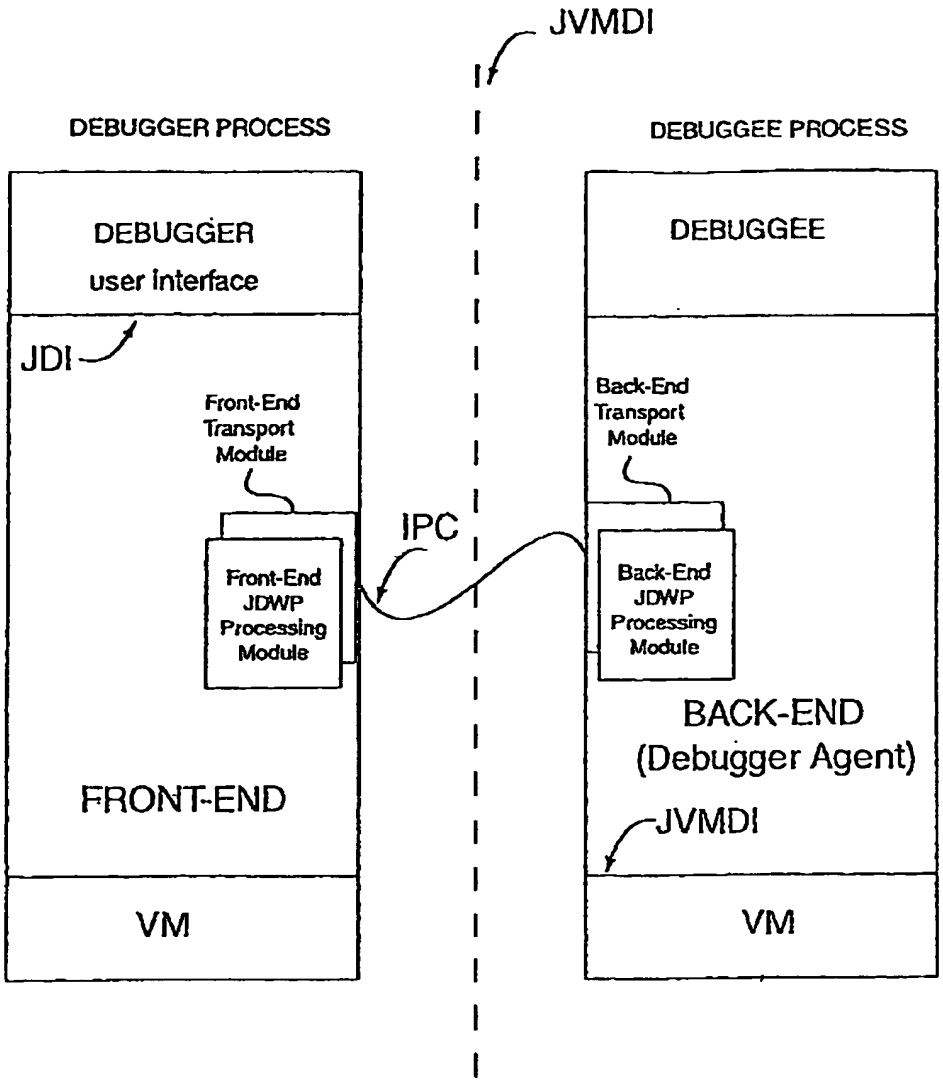
JVMDI
DEBUGGER PROCESS
DEBUGGEE PROCESS
DEBUGGER
user interface
JDI
Front-End
Transport
Module
Front-End
JDWP
Processing
Module
IPC
FRONT-END
VM
DEBUGGEE
Back-End
Transport
Module
Back-End
JDWP
Processing
Module
BACK-END
(Debugger Agent)
JVMDI
VM

Figure 2b

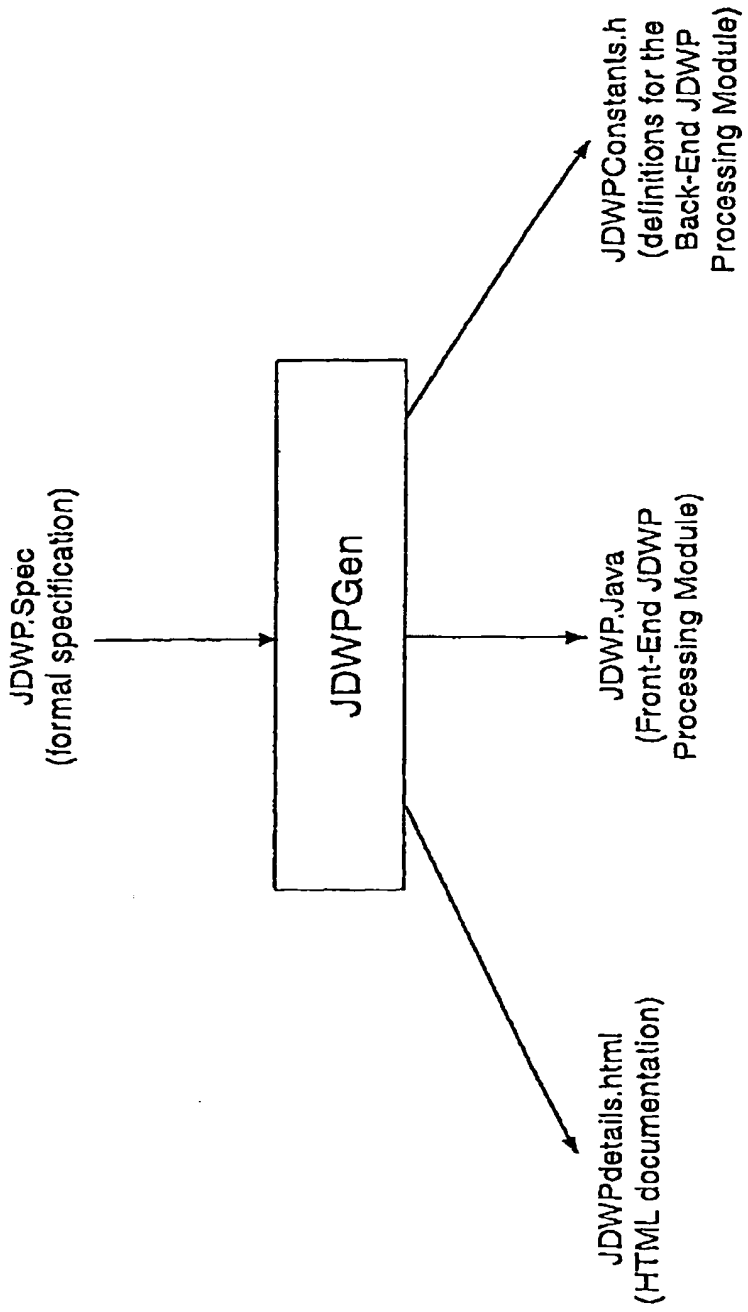
JDWP.Spec
(formal specification)
JDWPGen
JDWPConstants.h
(definitions for the
Back-End JDWP
Processing Module)
JDWP.Java
(Front-End JDWP
Processing Module)
JDWPdetails.html
(HTML documentation)

FIG. 3

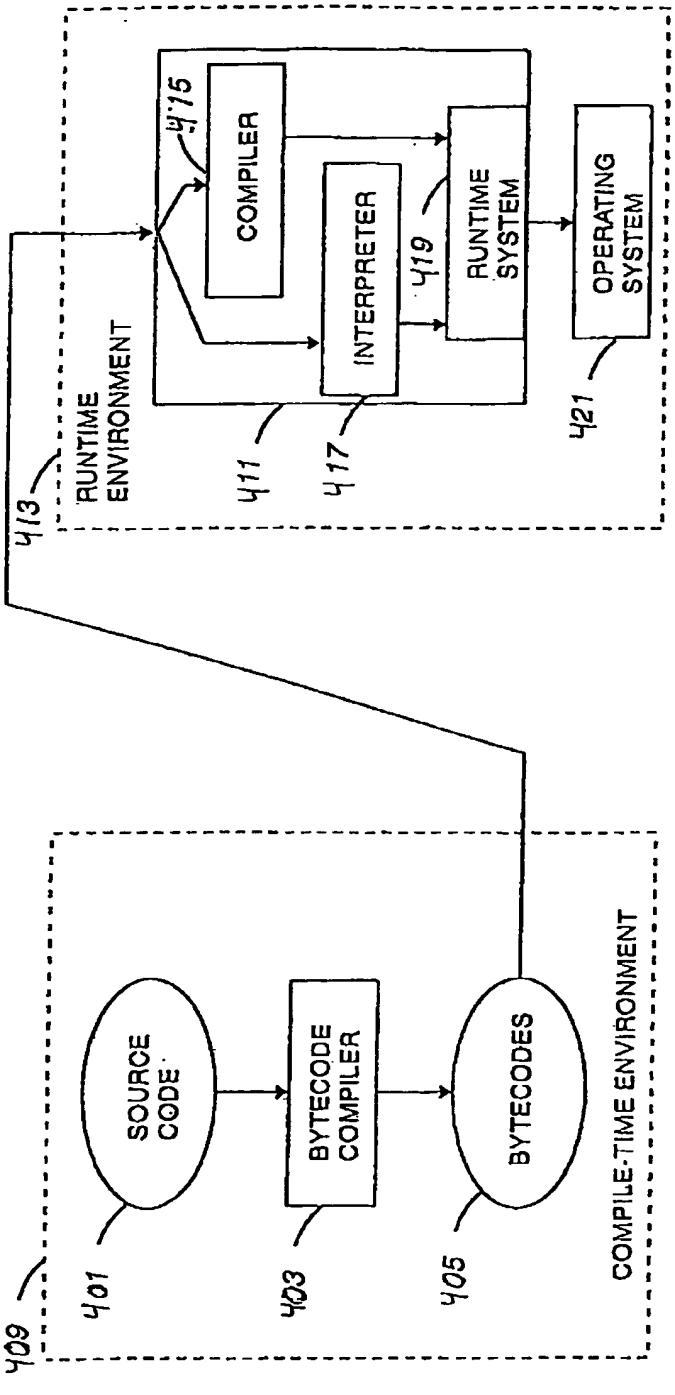
409
401
SOURCE CODE
403
BYTECODE COMPILER
405
BYTECODES
COMPILE-TIME ENVIRONMENT
413
RUNTIME ENVIRONMENT
411
415
COMPILER
417
INTERPRETER
419
RUNTIME SYSTEM
421
OPERATING SYSTEM

FIG.4

1. Abstract

A method for automatically generating front-end code and back-end code that are both compatible with a specification, such as the JDWP communication protocol. First, a detailed protocol specification is written that contains a description of an communication protocol between the front-end code and the back-end code. The detailed specification is then input into a code generator that parses the specification. The front-end code is then automatically generated from the formal specification, and may be written in a first computer language such as the Java(trademark) programming language. The code generator then generates the back-end code, which may be written in a second computer language such as C.

2. Representative Drawing

Fig.2a